CSE291-14: The Number Field Sieve

https://cseweb.ucsd.edu/classes/wi22/cse291-14

Emmanuel Thomé

February 8, 2022

Part 5b

Collecting relations in NFS

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Various sieving strategies

Depending on p^k, we will use varying strategies:

- for very small p, we will use pattern-sieving;
- mildly larger primes (say up to 2^I) are done with line sieving ("small sieve" in Cado-NFS).
- even larger primes are sieved with bucket sieving, which can even be done in several stages.

Plan

Sieving strategies

- Pattern sieving
- Line sieving (“small sieve”)
- Dividing into small regions

Pattern sieving

Idea. When p is tiny, there are many hits, and those are very close to each other.

We can use the fact that a processor likes to work with (64-bit) machine words rather than single bytes.

Used in Cado-NFS for $p = 2, 3, 5, 7$ and small powers.

Pattern sieving for 2 and its powers

For powers of 2:

- Prepare a block of 16 bytes.
- From the factor base data, we know where we should subtract $k \log 2$, for $k = 1, 2, 3$. Store this info in the block.
- Subtract this block from the sieving space, 2 unsigned long at a time.
- Alignment of the pattern changes with j.

Rem. No carries! Guaranteed! We do an `unsigned long`-level subtraction to emulate 8 independent byte subtractions.

Pattern sieving for 3

For 3:

- Still want to use unsigned longs. But the pattern must have a number of bytes that is divisible by 3.
- Use a block of 3 unsigned longs.
- Prepare and apply block as before.
- Alignment of the pattern changes with j.

More pattern sieving

Cado-NFS has more pattern sieving:

- Use SSE2 / AVX / . . . to apply the pattern.
- Handle larger primes (3,5,7).
- Instead of doing the pattern separately for each prime, use just one longer pattern (like in wheel sieve).

Pros/Cons

Obviously, pattern sieving is only suitable for (very) small primes.

Plan

Sieving strategies

- Pattern sieving
- Line sieving (“small sieve”)
- Dividing into small regions

Line sieving ("small sieve")

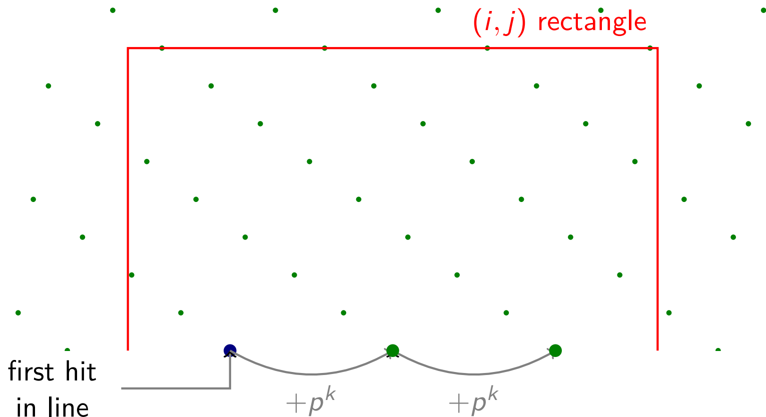
Warning. We use the terminology "line sieving" for a part of the lattice sieving. Do not forget we are still in a special-$\mathfrak{q}$ sublattice.

Context. This is used when $p^k < 2^I$.

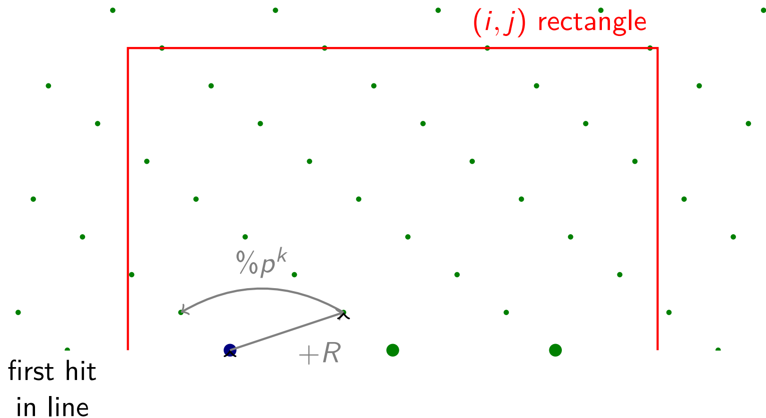
There is at least one hit per line of the (i, j)-plane.

Handle the j-lines one after the other (hence the name!).

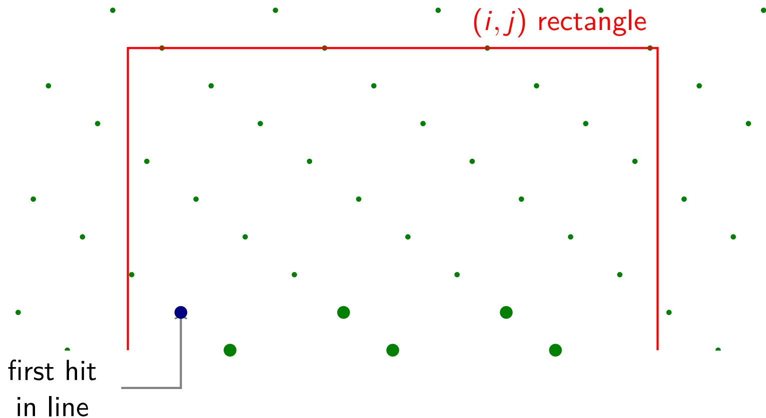
Line sieving ("small sieve")



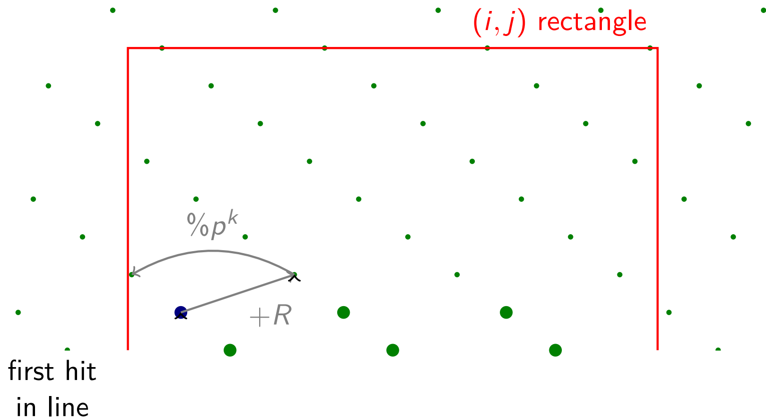
Line sieving (“small sieve”)



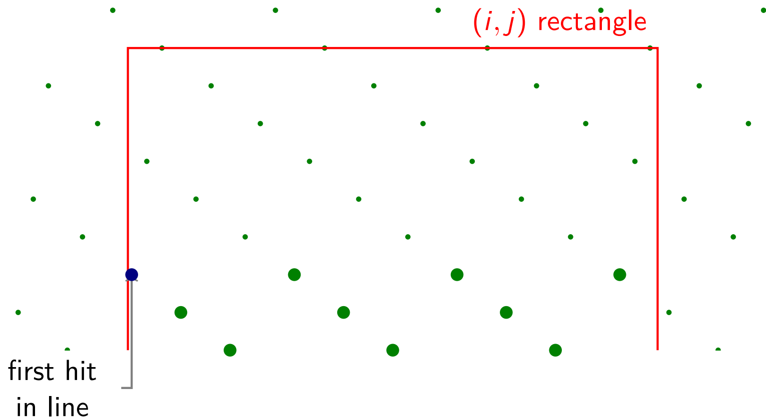
Line sieving ("small sieve")



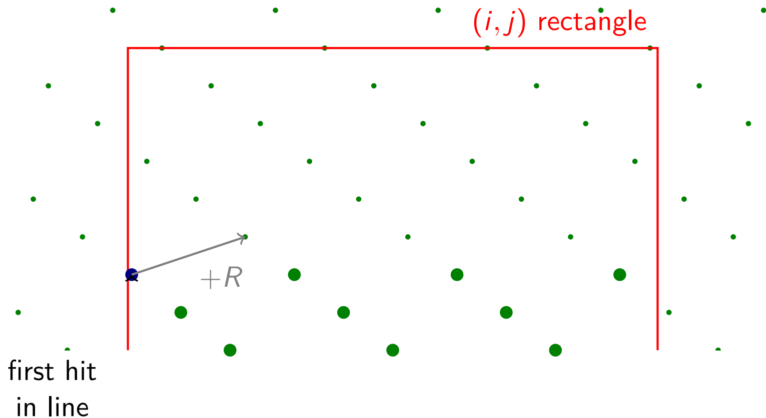
Line sieving ("small sieve")



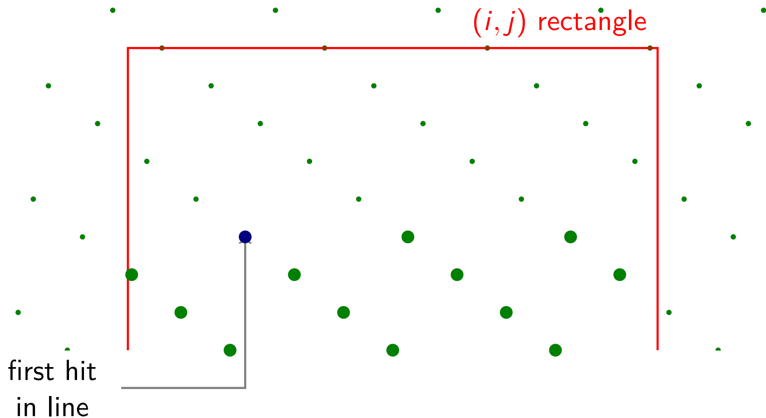
Line sieving ("small sieve")



Line sieving (“small sieve”)



Line sieving ("small sieve")



Line sieving

1. Initialize: $j = 0$, $t = -2^{I-1} + (2^{I-1} \bmod p^k)$.
2. *[work on the current line]*
3. set $u = t$ *[first hit in line]*
4. while $u < 2^{I-1}$,
5. subtract $\log p$ at position (u, j);
6. $u \leftarrow u + p^k$
7. Increment j, stop if $j \geq J$.
8. $t \leftarrow t + R$ *[first or maybe second hit]*
9. if $t \geq p^k$, then $t \leftarrow t - p$ *[first hit, really]*
10. Go to step 2.

Line sieving

Tricks:

- For j even, we sieve only odd i (and add $2p^k$ instead of p^k).
- In the projective case (when $p \mid j$ is a necessary condition), it makes sense to adapt the algorithm, and not go through lines which have assuredly no hit.
  Note that this is a rare case, and it is not a problem if it is dealt with by special code.
- For a given small p^k, since there are many hits, we have some positive aspects implementation-wise:
  - Computation time is spent in the tightest loop.
  - Memory accesses are close to each other.

Plan

Sieving strategies

- Pattern sieving
- Line sieving ("small sieve")
- Dividing into small regions

The sieve area is too large

Problem: The sieve area $2^A = 2^I \times J$ is often large.

If we line-sieve factor base primes for the whole (i, j) rectangle, then despite the good things that we mentioned, we have bad cache behaviour because we will traverse a multi-GB memory area over and over again.

Full memory traversal is expensive: it causes all memory caches to be emptied and refilled constantly!

The sieve area is too large

Dividing into small regions

It is better to split the sieve area in pieces.

- we typically handle sub-areas of the (i, j) rectangle 64 KB at a time.
  - Index within such an area can be 16 bits.
  - Always fits in L2 cache.
  - L1 cache is often smaller, though.
- our hope is that this will minimize cache misses.

As I grows, line sieving only on regions of 64 KB at a time means:

- we deal with only a fixed number of lines at a time.
- Extreme cases:
  - 1 line when $I = 16$
  - line fragments when $I > 16$!

Sieving regions: doing it right

Challenge: for each new region:

- we must resume all small sieve computation where we left them.
- this means: store the "first hit in line" as it would have been deduced, had we kept going.

This is reasonable as long as we have something to do with line-sieved primes in each region.

Clearly, if $p^k \geq$ region size, we do not want to go through all this.

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

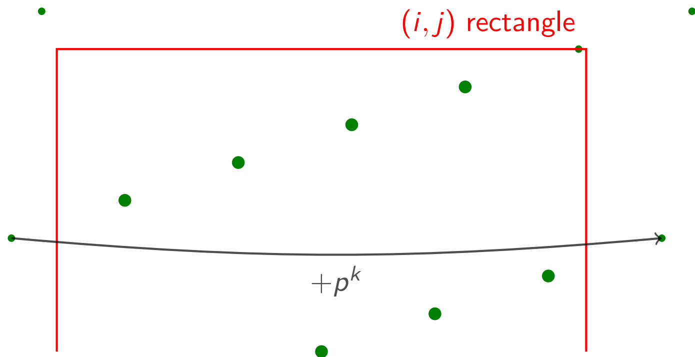
Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Many misses

Problem: when $p^k > 2^I$, there are lines with no hit.



Worse: when $p^k \gg 2^I$, almost no line has a hit. For p^k larger than the region size, not even every region has a hit.

A new enumeration for primes above 2^I

Idea

Because p^k is larger than the width of the (i, j) rectangle, there might be a way to enumerate all hits efficiently.

The answer is the Franke–Kleinjung algorithm:

- Goal: obtain a basis of the lattice that is well adapted to enumerating the hits.
- We intend to run that in a tight loop, and infer the list of (possibly rare) regions where hits will occur.

Note: most of the following description assumes $k = 1$. The case $k > 1$ for such p is only WIP in Cado-NFS anyway at the moment, and is not sure to be worthwhile.

Franke–Kleinjung's lattice sieving

Idea. Adapt the stopping criterion in Gauss algorithm to get a basis adapted to the width 2^I of the search space.

Lemma

There exists a basis $\langle v_0 = (\alpha, \beta), v_1 = (\gamma, \delta)\rangle$ of $\mathcal{L}_\mathfrak{p}$ such that

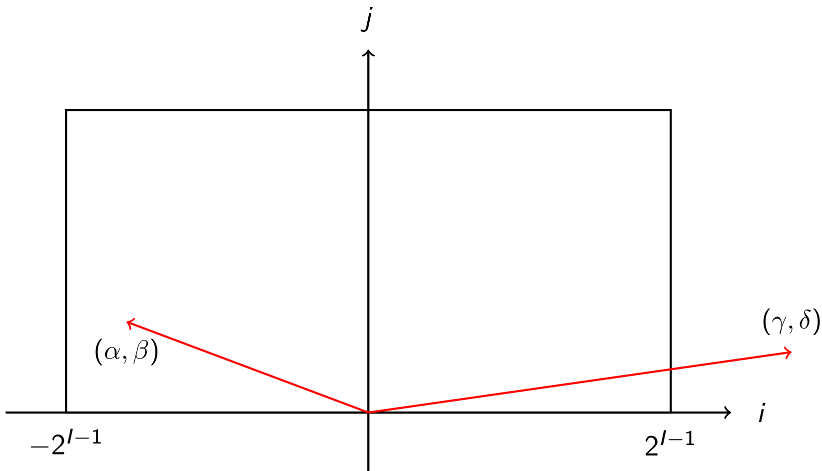
- β *and* δ *are positive;*
- $-2^I < \alpha \leq 0 \leq \gamma < 2^I$;
- $\gamma - \alpha \geq 2^I$.

The proof is neither really complicated, nor enlightening.

Implementation.

- Easy to get something that works; similar to Euclidean algorithm;
- Enumerating is costly because we have many primes: many hits to process.

Franke–Kleinjung's lattice sieving



Franke–Kleinjung's lattice sieving

Fact. This basis allows to iterate on the points of $\mathcal{L}_\mathfrak{p}$ that are exactly in the (i, j)-rectangle with no branching.

Let (i, j) be a valid hit. At most one of the following vectors is valid:

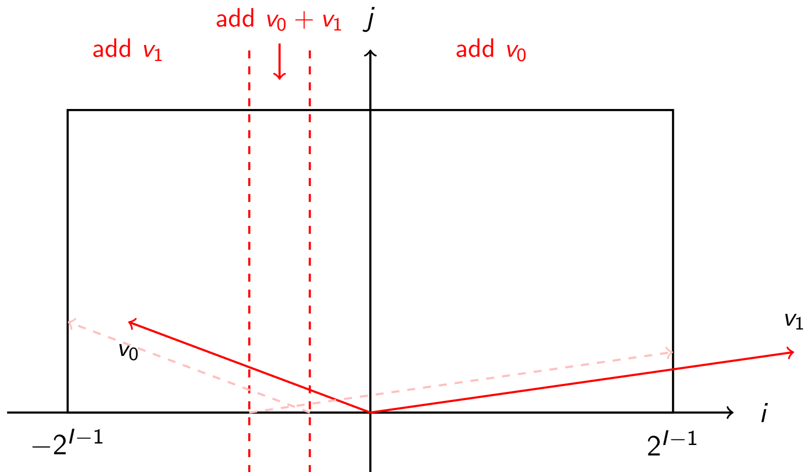
- $(i, j) + (\alpha, \beta)$;
- $(i, j) + (\gamma, \delta)$;

And if the two above vectors are invalid, then the following is valid:

- $(i, j) + (\alpha, \beta) + (\gamma, \delta)$.

Fact. Applying this rule starting with $(0, 0)$, we enumerate all points of $\mathcal{L}_\mathfrak{p}$ in the (i, j) rectangle.

Franke–Kleinjung's lattice sieving



Franke–Kleinjung's lattice sieving

Tricks:

- Deciding which vectors to add depends on easy bounds on i.
- This can be implemented in a branch-free way using `cmov`'s (nowadays, the C compiler does it for you).
- Assume that the memory layout is such that $T[(i,j)]$ is at address $j \cdot 2^I + (i - i_{\min})$. Then adding a vector to an index to the array T is done by adding an integer.

Handling the memory locality question

Problem. When $\mathfrak{p}$ is large, each hit in the (i,j)-rectangle is a cache miss.

If not careful, a lot of time is spent fetching remote memory into cache in order to subtract $\log p$ in the appropriate cell.

Think different!

- In term of information, the data produced by the sieve is a (huge) set of sieve updates, i.e. pairs (location, contribution).
- Instead of using directly the sieve update, store them for future use.
- Before applying the updates to the sieve array, sort them according to increasing locations.
- Then, applying the $\log p$'s contributions will be cache-compliant.

Handling the memory locality question

But. Sorting is not really local, is it?

- Sorting is a very well studied topic, and there exist variants that are local.
- For instance, `merge_sort` is quasi-linear even on a Turing machine.
- Even better: we do not need a perfect sorting, since at the L1 cache level, we can consider to have random access.

Solution: Bucket sorting.

Bucket sieving

We reuse our division of the (i, j) rectangle into regions.

Def. A region is a set of contiguous j-lines of the (i, j)-rectangle that fits (more or less) in the L1 cache.

The sieve array is therefore split in many regions.

- Prepare k buckets, i.e. storage for lists of sieve updates, each bucket corresponding to a region.
- Run the sieving à la Franke–Kleinjung for each $\mathfrak{p}$ in the factor base; for each hit, append the sieve update in the appropriate bucket.
- For each region, apply all the updates of the corresponding bucket.

Big steps of bucket sieving

1. Allocate buckets
2. **Fill buckets**: for both sides, and for all prime ideals above the bucket threshold,
   - Iterate through locations of updates with the Franke–Kleinjung enumeration.
   - Append each update in the right bucket (lower bits of memory location and some info related to p).
3. Loop over all regions (64 KB each):
   - Initialize (log-)norms.
   - **Apply buckets**: read locations from buckets, do subtractions.
   - do the line sieving for the small primes (both sides), for this region only.
   - Look for survivors that deserve further investigation.

Bucket sieving

Analysis.

When sieving, we have k pointers (one for each bucket) advancing in parallel.

If the cache can handle k cache lines, there are no misses.

Furthermore, processors tend to be optimized for linear memory access: we can hope for automatic prefetching.

Caveat.

Ever heard about TLB? (Translation look-aside buffer)

This is a key element of the virtual memory mechanism. One can think of it as a cache for the big table that contains the correspondence physical / logical addresses.

If the number of buckets k is larger than the TLB, we often have TLB misses.

Bucket sieving — details

Bucket updates:

- Buckets take a lot of memory (but then we do not need all the sieving area in memory).
- Do not need to store the high bits of j ($=$ bucket id).
- Do not need to store $\log p$. Indeed, primes are sieved in increasing order, so we can just remember the few events when $\log p$ increases in a bucket.
- For re-sieving (later), it would be nice to have p, but store only 16 bits of p, called a hint.
- Conclusion. Only 32 bits per update.

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Recap

We have seen so far:

- How we sieve, for tiny, for small, and also for larger primes.
- The (i, j) sieving rectangle is divided into regions, typically of 64 KB.
- The bucket sieving process computes in advance bucket updates. Buckets are attached to regions.

Recap

We thus have the bits and pieces for an algorithm that identifies (i, j) coordinates for which...

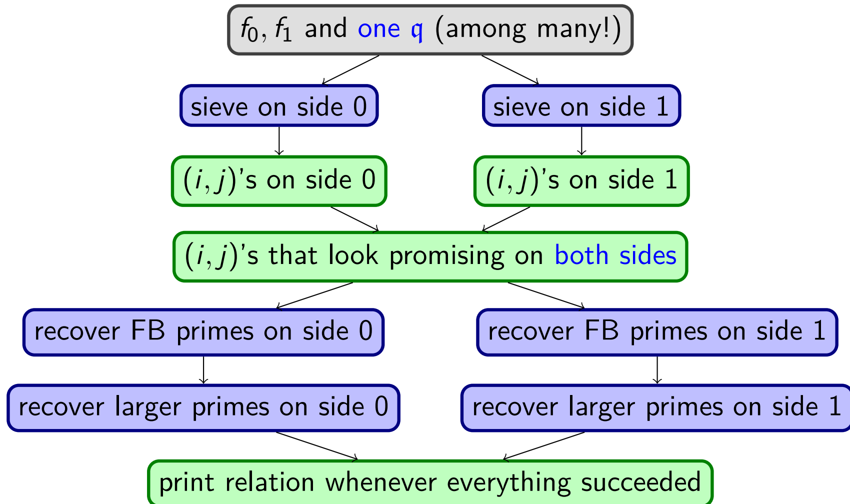
- $\text{Res}(a - bx, f_0(x))$ and $\text{Res}(a - bx, f_1(x))$ are both smooth;
- or at least, both have a large contribution from FB primes. In this last case the norms might be almost smooth, and it might well be that this is good enough for our purposes.

Recap

Next: we need to recover the complete factorization for all these promising locations (i, j).

- Some factors come from the factor base: some work is needed to find them again: re-sieving.
- When norms are just promising, not completely smooth, cofactorization will be used to find the remaining factors.

Graphically



Parameters

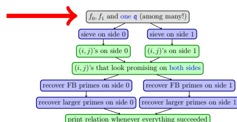
Cado-NFS's lattice siever is called `las`.

The `las` program is quite versatile, and is controlled by a host of (command-line) parameters.

Almost every transition in the previous slide can be controlled by a parameter.

Parameters: the special-$\mathfrak{q}$'s

Several parameters can be used to select $\mathfrak{q}$'s.



First: `sqside` is (integer) side 0 or 1.

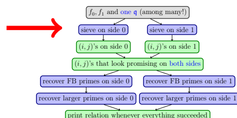
`q0`, `q1`: all special-$\mathfrak{q}$ in the range $q \in [q_0, q_1)$.

`q0`, `q1`, and `-random-sample n`:
only n special-$\mathfrak{q}$'s, evenly spaced across $[q_0, q_1)$.

`q0`, `rho`: just one single special-$\mathfrak{q}$, the ideal $(q_0, x - \rho)$.

`las` also agrees to work with an arbitrary list of special-$\mathfrak{q}$'s given by the `todo` parameter.

Parameters: the sieving parameters



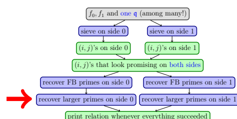
`I` or `A`: size of the (i, j) sieving area.

`lim0`, `lim1`: factor base bounds on both sides.
We sieve for $p <$ `lim0` on side 0.

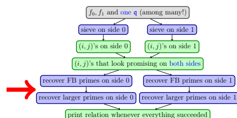
`bkthresh`: when we start to use Franke–Kleinjung enumeration (a good default value is 2^I);

`tdthresh`, `bkthresh1`, `adjust-strategy`: more advanced parameters.

Parameters: large primes

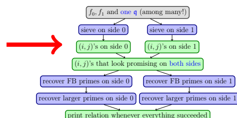


`lpb0`, `lpb1`: bit size of the largest primes that we tolerate in relations.



`mfb0`, `mfb1`: maximum bit size of norms after removing all sieved primes.

Parameters: after-sieve threshold



`lambda0`, `lambda1`: coarse-grain test.

We would like to keep (i, j) whenever the unsieved part is below `mfb` on both sides.

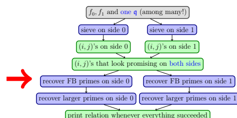
- However we do not know the bit size of the unsieved part, just a coarse approximation of it.
- We compare this approximation with `lambda` $\times$ `lpb`. `lambda` defaults to `mfb`/`lpb`, but it is allowed to put explicit values.

Re-sieving

If a cell passes the λ-threshold on both sides, we need to compute the corresponding norms and test them for smoothness according to the large prime bound: need exact cofactor.

- Norm computation: not so much tricks to do, since we need an exact result.
- We handle a region at once and deal with all the survivors: we can therefore do again the line sieving for the small primes (and divide the norm by p).
- The buckets are also applied again, but this time the primes are reconstructed from the hints and trial divided in the norm.
- The remaining part of the norms are tested against the exact threshold, and possibly factored with ECM (cofactorization).

Parameters: trial division



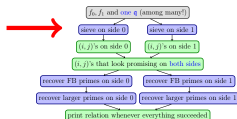
`tdthresh`:

For the smallest primes, divide them out at the promising (i, j) positions by trial division only.

Rationale: when p is really small, it is quick enough to trial-divide all the survivors.

Wait, we have two sides!

The activity on both sides is interleaved.



1. Fill the buckets on side 0;
2. Fill the buckets on side 1;
3. Loop on each region:
4. Initialize norms on side 0;
5. Pattern- and line- sieve on side 0;
6. Apply bucket updates on side 0;
7. Initialize norms on side 1 (only survivors);
8. Pattern- and line- sieve on side 1;
9. Apply bucket updates on side 1;
10. Cofactorize both sides (includes re-sieving).

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

How do we parallelize?

1. Allocate buckets
2. **Fill buckets**: for both sides, and all (large-ish) prime ideals

   ...
3. Loop over all regions (64 KB each):
   - **Apply buckets**
   - and do very local things on with this 64 KB region.

Parallelize in "two directions"

We parallelize steps 1 and 2 in one direction, and step 3 in another.

Multi-threading

This way of organizing the computation is "easy" to parallelize in a shared-memory environment.

- Steps 1 and 2 (allocate / fill buckets): each thread has its own set of buckets and takes care of part of the factor base.
- Thereafter, all buckets are readable by everyone.
- Step 3: Region are processed independently by all threads.
  - Region r is processed by thread $r \bmod T$ (if T threads).

Advantage: The memory pressure per core is reduced.

Splitting the factor base

For steps 1 and 2, we want to split the factor base in many “slices” so that, for the overhwelming majority of them:

- slices contain roughly an equal number of prime ideals.
- slices contain no more than 2^{16} prime ideals.
- all prime ideals in a slice are easy ideals, and not projective roots in the (i, j) plane.
- all prime ideals in a slice have equal $\lfloor \log_\beta |\mathrm{Norm}(\mathfrak{p})| \rceil$.
- all prime ideals in a slice are above prime number with equal number of roots modulo p.

These restrictions are here so that the inner loop can be streamlined as much as possible.

Processing regions in parallel

Thread k deals with regions $k, k + T, \ldots$.

- To make the "small sieve" efficient, the per-line first hits for each prime must be remembered from one region to the next. Some annoying arithmetic adjustments are necessary.
- It gets worse when lines are larger than one region.
- More pragmatic approach: precompute the starting points for all "small-sieved" primes and all regions. That's cheap enough.

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Memory footprint

With bucket sieving, most of the memory goes into the storage of bucket updates.

$$\text{number of updates} \approx \#\mathcal{A} \times \sum_{\substack{\mathfrak{p}\in\mathcal{F}_0 \\ \mathfrak{p}\ \text{bucket-sieved}}} \frac{1}{|\operatorname{Norm}\mathfrak{p}|}.$$

$$\approx \#\mathcal{A} \times (\log\log \mathtt{lim} - \log\log \mathtt{bkthresh}).$$

Orders of magnitude:

- $\#\mathcal{A} \approx 2^{34}$. `lim` around 2^{32}.
- $I \approx 16$ and $\mathtt{bkthresh} = 2^I$.
- $\log\log 2^{32} - \log\log 2^{16} \approx 0.69$
- 4 bytes per update. Total about 13G.

Too many buckets at a time

When $\#\mathcal{A} \approx 2^{34}$ and regions are 64 KB:

- That makes about 2^{18} buckets that we are writing to.
- (way) more than the CPU can handle efficiently.

We can alleviate both problems at once with multi-layer bucket sieving.

Multi-layer

Simple idea (but not trivial implementation-wise):

- For primes above some threshold (say about 2^{I+8}), dispatch into "big buckets" that correspond to "big regions", equivalent to 2^8 regions at a time.

  > Cado-NFS calls this threshold `bkthresh1`.

- Subdivide the region processing into blocks 2^8 regions. When we begin such a block:
  - we bucket-sieve the primes between `bkthresh` and `bkthresh1`.
  - and do a secondary pass on the updates from the current "big bucket".

How do we handle prime ideals?

- Largest factor base prime ideals.
  - $\#\mathcal{A} \times (\log\log \texttt{lim} - \log\log \texttt{bkthresh1})$ updates in total.
  - Dispatched at the beginning of sieving into $\lceil \#\mathcal{A}/2^{24} \rceil$ "big buckets".
  - Re-dispatched in a second pass when we begin processing the first region in a batch of 2^8.
- Prime ideals between `bkthresh` and `bkthresh1`.
  - Dispatched among the "next 2^8 regions" only when we're about to process them.
  - This is fine, since these have many hits anyway.
  - Need memory for $2^{24} (\log\log \texttt{bkthresh1} - \log\log \texttt{bkthresh})$ updates.
- Primes below 2^I are line-sieved only when we process a region.

Pros/Cons

Two advantages:

- We limit the number of buckets that we are writing to.
- The memory cost is reduced:

$$\text{number of updates} \approx \#\mathcal{A} \times (\log\log \texttt{lim} - \log\log \texttt{bkthresh1}) \\ + 2^{24} (\log\log \texttt{bkthresh1} - \log\log \texttt{bkthresh}) .$$

Downside: the processing of the large primes requires two steps.

Two-layer bucket sieving matters a lot in large computations.

- it can probably be improved: the current "hard" cutoffs are not ideal.
- it might make sense to have three layers at some point.

Plan

Sieving strategies

Bucket sieving and the Franke–Kleinjung enumeration

General organization of the siever, and main parameters

Multithreading and more implementation details

Multi-layer bucket sieving

Using sieving AND batch smoothness detection

Use case

Two fundamental differences between time and memory requirements of sieving and batch smoothness detection.

- Sieving: dependence on $\#\mathcal{A}$.
- Batch smoothness detection: dependence on the number of integers we want to test for smoothness.

Since NFS sieving has two sides, it makes sense to:

- Do sieving on the "hard" side.
- Once the rare survivors are identified, feed them to batch smoothness detection.
- Batch smoothness detection can even run asynchronously: do a dozen special-q's, then do batch smoothness detection on the survivors, etc.

Part 5c

Collecting relations in NFS: two further topics

Norm initialization

Adjusting the $\mathfrak{q}$-lattice

Plan

Norm initialization

Adjusting the $\mathfrak{q}$-lattice

Norms: what is it about?

Context

- We have the polynomials f and g.
- We have chosen a special-$\mathfrak{q}$ on one of the two sides.
  So we are working with

$$(a, b) = (i \cdot (a_0, b_0) + j \cdot (a_1, b_1))$$

  for (i, j) in the (i, j) rectangle.
- We will sieve on both sides because we want both $|\mathrm{Res}(a - bx, f(x))|$ and $|\mathrm{Res}(a - bx, g(x))|$ to be smooth.

Sooner or later we will need to compare two things

- The accumulated contribution of sieved factor base primes p;
- and the size of the $|\mathrm{Res}(a - bx, f(x))|$ (on the side we're sieving on).

Norms: what is it about?

- The accumulated contribution of sieved factor base primes p;
- vs: the size of the $|\mathrm{Res}(a - bx, f(x))|$.

We do this comparison additively

- at the beginning of the computation, we store $\log|\mathrm{Res}(a - bx, f(x))|$ at the location corresponding to (i, j).
- each time we identify a location where p divides the norm, we subtract $\log p$ to that location.

We choose the log base so that the computation can be done using single-byte integer arithmetic.

This inaccuracy is an acceptable trade-off.

Norms: what is it about?

Each cell contains an 8-bit integer:

- Initialized with the logarithm of the norm.
- During sieving, the logarithm of the prime to be divided is subtracted.
- After sieving, positions where the cell contains a small integer are promising: only consider those for cofactorization.

After sieving, the cell corresponding to (i, j) contains an approximation of the log of the part of $\mathrm{Res}(a - bx, f(x))$ that is made of primes that were not sieved.

In NFS, depending on this log value, we may decide to:

- strive to factor them (e.g. with ECM);
- or ignore them because they stand too little chance of being interesting in the end.

Norm initialization in Cado-NFS (I)

When developing an NFS sieving program, it is important to be precise with norm computations at the beginning, because this is an important debugging asset.

Early versions of Cado-NFS:

- precise (but slowish) computation of norms;
- various tricks come into play;
- computation of algebraic norms more expensive;
- still claims a share of the total computation cost that is way too large (10%).

Norm initialization in Cado-NFS (II)

More recent improvements in the norm computation (version 2.1, improved in 2.3):

- Unify algebraic and rational norm initialization by computing piecewise linear approximations of polynomials that are accurate up to a multiplicative factor.
- Neighbouring cells often have the same value. We compute the value changes instead (easy for a linear polynomial).

Piecewise linear approximations

Input:

- a polynomial $\tilde{f}(x)$ with real coefficients. $\tilde{f}(x)$ is such that

$$\tilde{F}(i,j) = j^{\deg \tilde{f}}\tilde{f}(i/j) = \mathrm{Res}(a - bx, f(x)).$$

  The $\mathfrak{q}$-lattice defines a homography that yields $\tilde{F}$ from F:

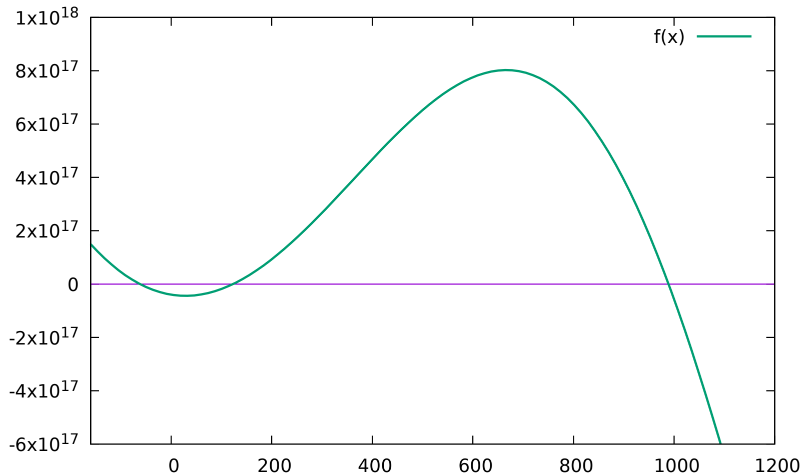
$$\tilde{F}(i,j) = F(ia_0 + ja_1, ib_0 + jb_1).$$

- an inaccuracy tolerance τ;
- a range of interest $[-2^{I-1}, 2^{I-1}] \times [1, J]$.

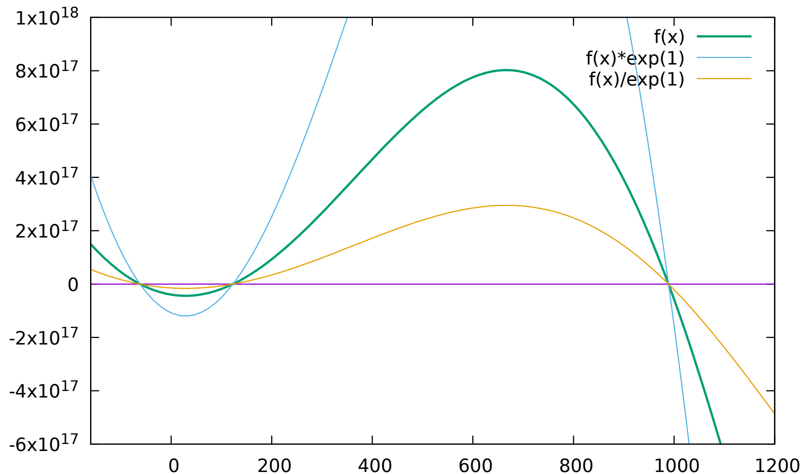
output: a list of linear functions $u_0, \ldots, u_{k-1}$ and consecutive intervals $R_0, \ldots, R_{k-1}$ such that $\cup_s R_s = [-2^{I-1}, 2^{I-1}]$ and

$$\forall s,\ \forall i \in R_s,\ e^{-\tau}|\tilde{F}(i,1)| \leq u_s(i) \leq e^{\tau}|\tilde{F}(i,1)|$$

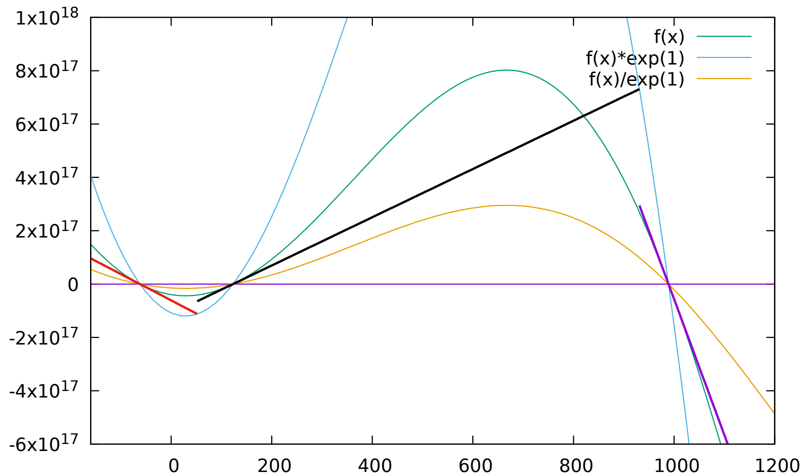
Piecewise linear approximations: example



Piecewise linear approximations: example



Piecewise linear approximations: example



Piecewise linear approximations

Once we have the PL approximations correct, then we can say that:

$$\left|\log|\tilde{F}(i,1)| - \log|\text{approximation}(i)|\right| \leq \tau;$$

$$\left|\log|\tilde{F}(i,j)| - j^{\deg f} \log|\text{approximation}(i/j)|\right| \leq \tau + (\deg f)\log j.$$

- it makes sense to compute PL approximations for more than just line $j = 1$.
- If we computed PL approximations for j_0, then the inaccuracy drops to $(\deg f)(\log j - \log j_0)$, so approximations on lines spaced in a geometric progression are fine.

(log)norm computation for linear polynomials

Input: $\tilde{u} = c_1 x + c_0$. Want to compute $\log_\beta |c_1 i + c_0 j|$ for some j, and for all i: $-2^{I-1} \leq i < 2^{I-1}$.

- compute real root $\zeta = -(c_0 j)/c_1$.
- set $i = i_{\min} = -2^{I-1}$.
- compute $y = \left\lfloor \log \tilde{U}(i,j) \right\rceil$.
- if $i < \zeta$, compute i' s.t. $\left\lfloor \log \tilde{U}(i',j) \right\rceil = y - 1$.
  (if $i > \zeta$, aim at $y + 1$ instead).
- fill the table with y until location i', and resume from there.

Finding i' is easy enough precisely because $\tilde{u}$ is a linear approximation.

Plan

Norm initialization

Adjusting the $\mathfrak{q}$-lattice

A foreword on J (integer bound)

Duh, some special-$\mathfrak{q}$'s are much faster than others:

```
# 3 relation(s) for side-1 (1310000579,873183740)
# Time for this special-q: 28.8720s [norm 0.0280+0.0200, [
    sieving 26.5920 (22.4760 + 1.0400 + 3.0760),        [
    factor 2.2320 (1.8160 + 0.4160)]
```

while for others:

```
# 19 relation(s) for side-1 (1310009947,283600118)
# Time for this special-q: 126.4920s [norm 0.1800+0.4480,
    sieving 111.3360 (86.7800 + 6.4040 + 18.1520),
    factor 14.5280 (11.7120 + 2.8160)]
```

Explanation comes from J.
(I'm not talking about the ideal J here!)

Lattice sieving

We sieve for $(a, b) = i \cdot \vec{u_0} + j \cdot \vec{u_1}$.

- $(\vec{u_0} = (a_0, b_0), \vec{u_1} = (a_1, b_1))$ basis of the lattice $\mathcal{L}_\mathfrak{q}$.
- We pick $\vec{u_0}$ the shortest vector in $\mathcal{L}_\mathfrak{q}$.
- We let typically $-I/2 \leq i < I/2$ and $0 \leq j < J$, have in mind $J = I/2$ too.
- We have $\det \mathcal{L}_\mathfrak{q} = q$.
  Furthermore, because of skewness, $a_i/b_i \approx s$.
- Bottom line: the vectors $\vec{u_i'} = (a_i\sqrt{\frac{1}{qs}}, b_i\sqrt{\frac{s}{q}})$ are relevant to plot.

Following slide: plot $[-I/2, I/2]\vec{u_0'} + [0, I/2]\vec{u_1'}$.

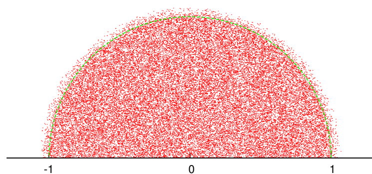
What happens?

It might really happen that the sieving rectangle looks pretty distorted.

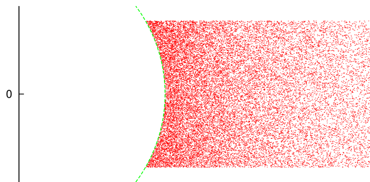
If $\vec{u_0}$ is really really short, then $\vec{u_1}$ is somewhat bigger, since the lattice determinant is constrained.

As $\mathfrak{q}$ varies, one may wonder how the quotient of complex numbers $\frac{u_1}{u_0}$ evolves. There are theorems for that.

A familiar shape



(a): u_0



(b): ratio $\tau = u_1/u_0$
(turn your head clockwise)

It's even possible to write down the density around $u_1/u_0 = x + iy$, which is $\frac{3}{\pi y^2}\mathrm{d}x\mathrm{d}y$.

So, bad stuff happens, sometimes.

Historical provision in `las` against that

Some straightforward options.

- Discard $\mathfrak{q}$ when it so happens that u_1/u_0 has large imaginary part (that is, when $\vec{u_0}$ is exceptionally small).
- Other option: limit J to a smaller value in that case.
  `las` does exactly this.

The issue we observe is that J is sometimes reduced a LOT.

Better strategy for adjusting I, J

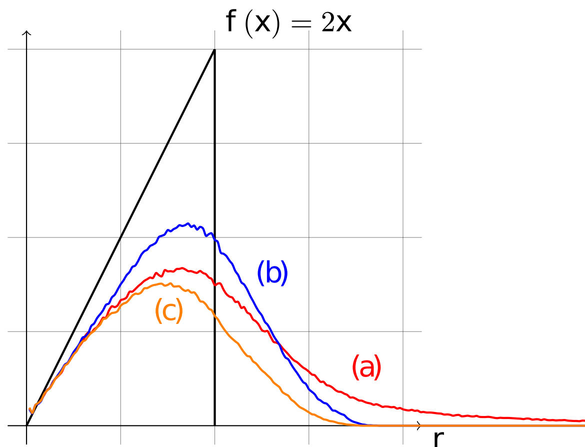
Assume we have in mind an area $A \approx 2^{31}$ within the (i,j) plane.

We sieve for $(a,b) = i \cdot \vec{u_0} + j \cdot \vec{u_1}$.

Obvious "we should rather do this" strategy.

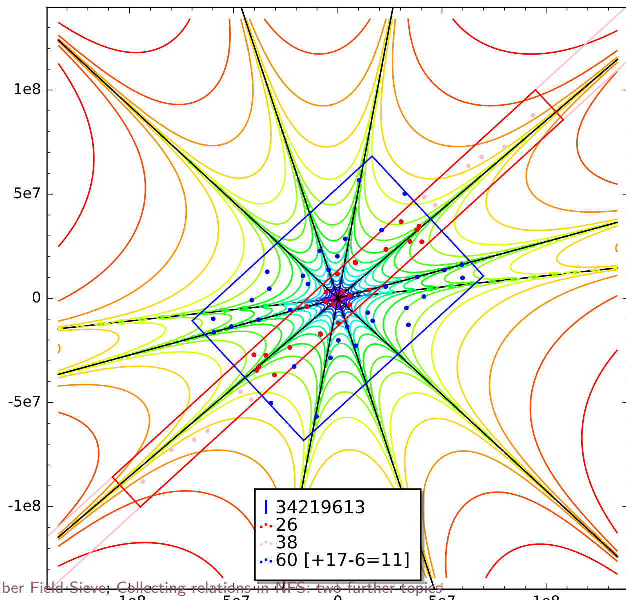
- Multiply the shortest vector by the largest interval.
- Ah, yes, but our I is limited to 2^{16}. (no longer in the development version — still at work!)
- One of the intervals, for sure, will be at least $\sqrt{A}$. Depending on u_1/u_0, we might prefer $\sqrt{A \cdot S}$ for some $S > 1$. The closest power of two is within $[\sqrt{AS/2}, \sqrt{2AS}]$.
- J, on the other hand, is not limited. So we should allow swapping vectors and reducing I instead for distorted lattices.

Relevant curve

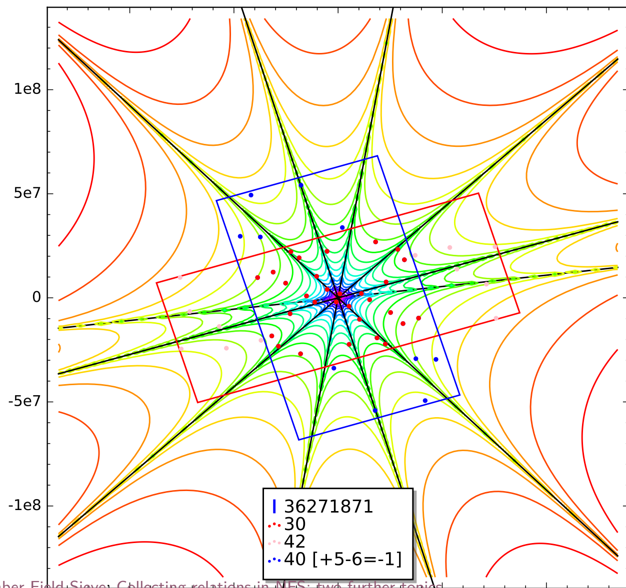


This is by no means a game changer, but will avoid wasting some special-$\mathfrak{q}$'s.

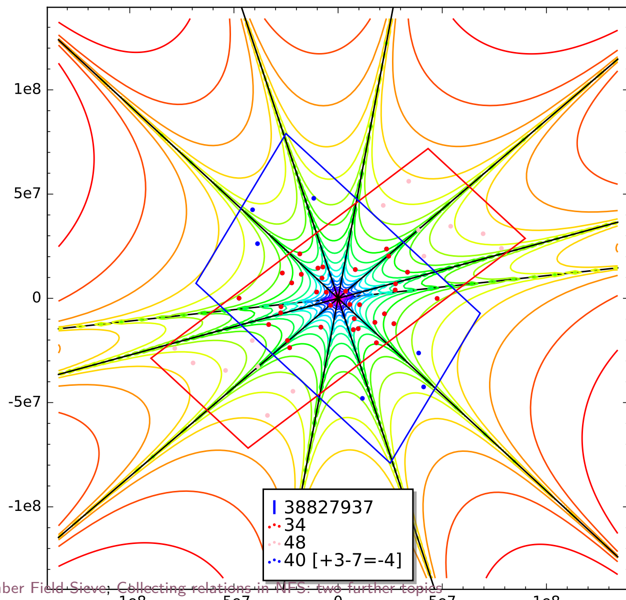
Some example plots



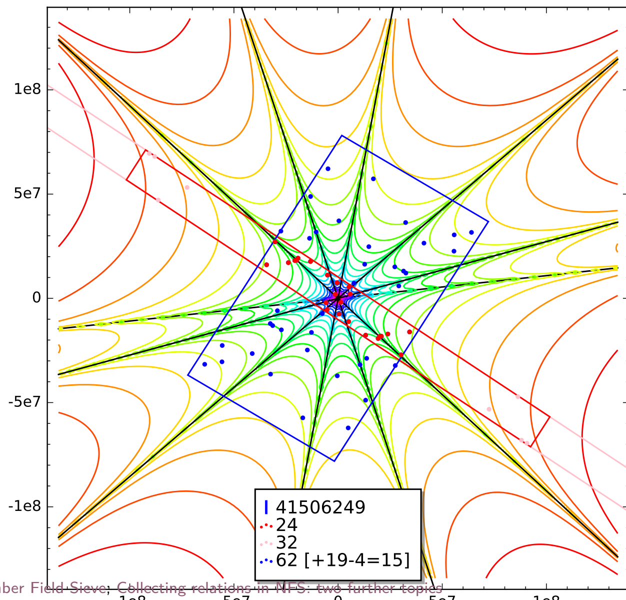
Some example plots



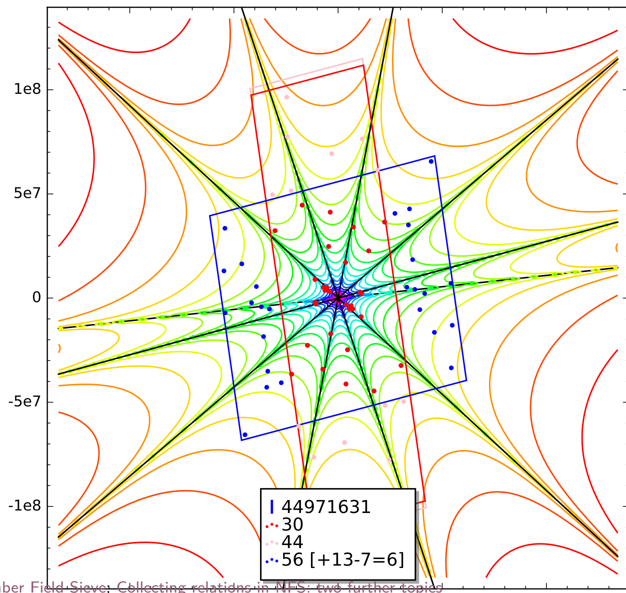
Some example plots



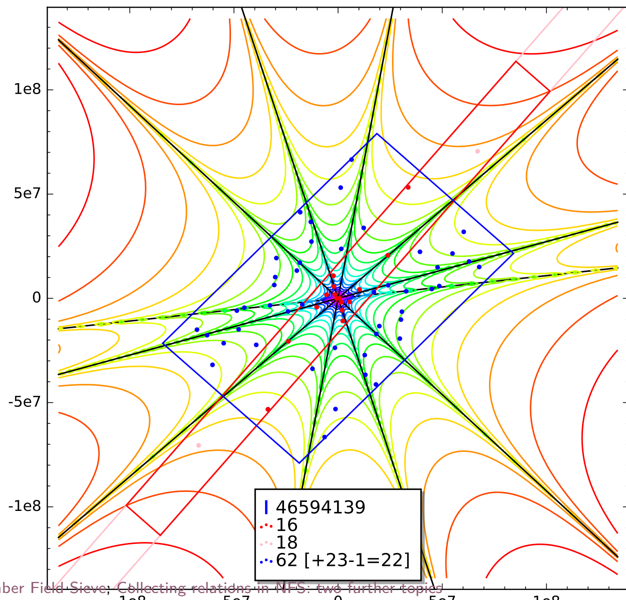
Some example plots



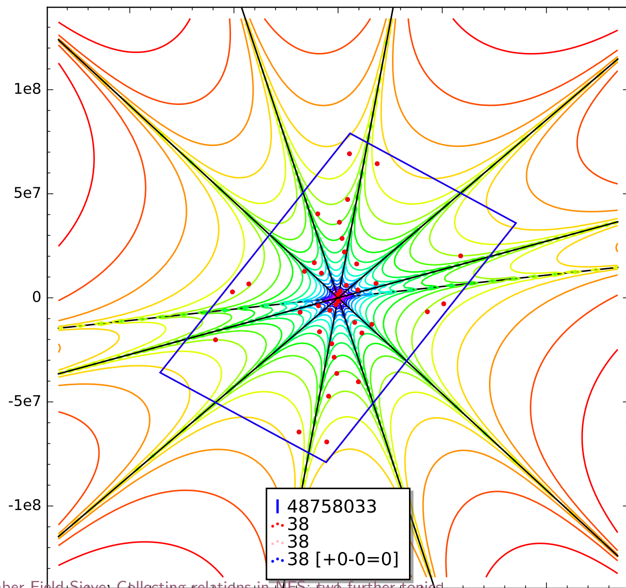
Some example plots



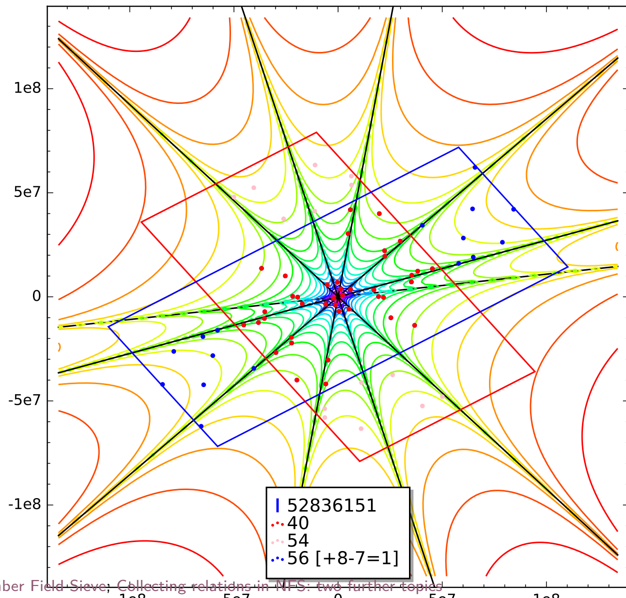
Some example plots



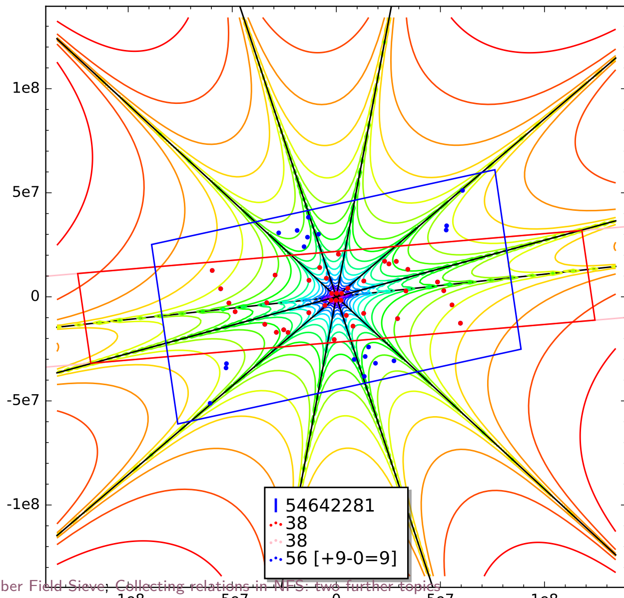
Some example plots



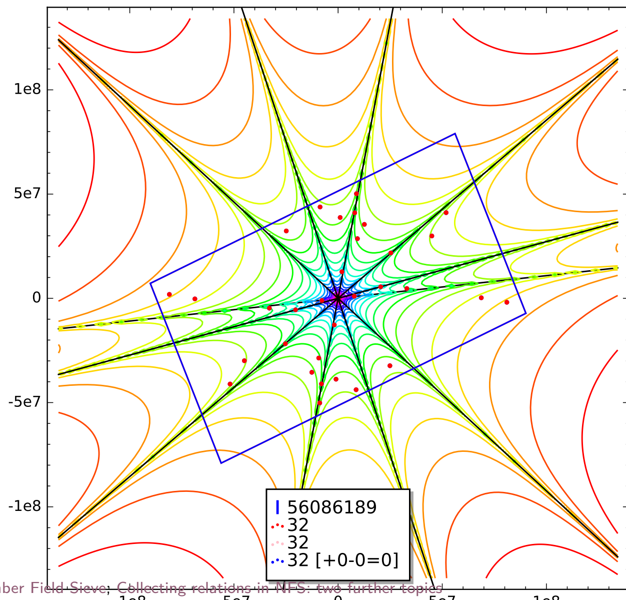
Some example plots



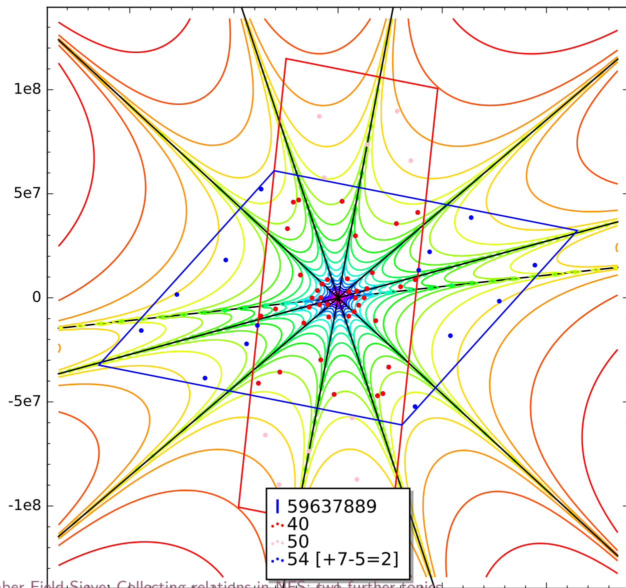
Some example plots



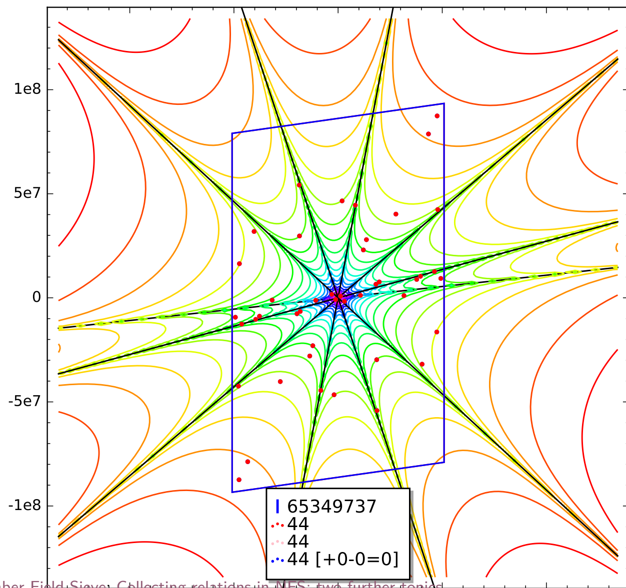
Some example plots



Some example plots



Some example plots



Some example plots