Integer points close to a transcendental curve... Dirty little tricks

Nicolas Brisebarre (C.N.R.S.) and Guillaume Hanrot (É.N.S. Lyon)

NuSCAP days in Toulouse - June 8, 2022

Outline

- Main optimizations ;
- The implementation(s) ;
- Experimental results.

Main optimizations (1)

- Use of Newton polynomials ; [theoretical]
- Efficient computation of DCTs ; [implementation]
- Prereduction ; [implementation]
- LLL truncation. [implementation]

Main optimizations (2) – Newton polynomials

Newton polynomials are $N_k(X) = X(X-1)\dots(X-k+1)/k!$; in 2D, we use $N_k(x)N_\ell(y)$.

- **Remark** : $N_k(\mathbb{Z}) \subset \mathbb{Z}$.
- Rather than looking for $P = \sum_{0 \leqslant u+v \leqslant d} \alpha_{uv} X^u Y^v$, we look for $P = \sum_{0 \leqslant u+v \leqslant d} \alpha_{uv} N_u(X) N_v(Y)$.
- Gives a slightly denser lattice...
- hence slightly shorter vectors...
- hence larger intervals...
- $\Rightarrow$ fewer intervals.

Main optimizations (3) – Strategy

- Main steps :
  - Coefficient matrix construction : mostly DCT ;
  - Remainder matrix construction : cheap ;
  - LLL ;
  - Checking size of vectors ;
  - Bivariate system solving (Hensel) ;
- Depending on parameters, the ratio matrix construction / LLL can vary a lot ;
- The other steps can be (and have been) agressively optimized.

Main optimizations (4) – Coefficient matrix

- Compared with Nicolas' talk: in real life, rather than use the values of f_{ij}, we use the DCT of f_{ij}.
- Try to work out a reasonably efficient DCT – use "fast DCT" ideas depending on N_1, N_2.
- Use relations between DCTs coming from $2xT_n(x) = T_{n+1}(x) + T_{n-1}(x)$. If $\delta =$DCT$(f(x))$, $\delta' =$ DCT$(xf(x))$ over $[a, b]$, we have
  - $\delta'[0] = (b-a)\delta[1]/4 + (b+a)\delta[0]/2$,
  - $\delta'[k] = (b-a)(\delta[k-1] + \delta[k+1])/4 + (b+a)\delta[k]/2$, $1 \leqslant k \leqslant n-2$,
  - $\delta'[n-1] = (b-a)\delta[n-2]/4 + (b+a)\delta[n-1]/2$.
- Only $d-1$ "actual" DCT associated to $f(x)^j$ have to be computed.

Main optimizations (5) – Prereduction

- Over $I = [a, b]$, we work with the matrix $F_{[a,b]}$ defined by

$$\begin{pmatrix} f_{0,0}(x_0) & \cdots & f_{0,0}(x_{N-1}) & r_{0,0} & 0 & \cdots & \cdots & 0 \\ f_{0,1}(x_0) & \cdots & f_{0,1}(x_{N-1}) & 0 & r_{0,1} & 0 & \cdots & 0 \\ \vdots & \cdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \vdots & & & & \\ f_{d-1,1}(x_0) & \cdots & f_{d-1,1}(x_{N-1}) & \vdots & \cdots & 0 & r_{d-1,1} & 0 \\ f_{d,0}(x_0) & \cdots & f_{d,0}(x_{N-1}) & 0 & \cdots & \cdots & 0 & r_{d,0} \end{pmatrix}$$

  and LLL gives $U = U_{[a,b]}$ such that $UF_{[a,b]}$ is small.
- Prereduction is hoping that $F' = U_{[a,b]}F_{[b,2b-a]}$ is small-ish – much smaller than $F_{[b,2b-a]}$.
- Then, LLL-reduce $F' \Rightarrow V$ st. VF' is small ;
- Put $U_{[b,2b-a]} = VU_{[a,b]}$.
- Experimentally, works pre tty well here ;
  - LLL becomes a lot cheaper ;
  - very efficient for large d.
  - Does not work for SLZ : $U_{[a,b]}F_{[b,2b-a]}$ is LARGER than $F_{[b,2b-a]}$

Main optimizations (6) – Prereduction

Very preliminary intuition :

- SLZ changes the problem $|y - f(x)| < \varepsilon$ to $|y - P(x)| < \varepsilon - \varepsilon'$, for P st. $|f - P| \leqslant \varepsilon - \varepsilon'$;
- ... then only works with P ;
- We study the problem $|y - f(x)| < \varepsilon$ but write $f = P + r$, for some r st. $|r| \leqslant \varepsilon'$;
- We work with $P + r = f$ all along ;
- Better connection for problems over neighouring intervals ;
- Heuristic arguments predict that the "work" left for LLL becomes $(3 + 2\sqrt{2})^N \approx 6^{d^2/2}$ vs 2^{dp} without prereduction.
- For $d = 12$, $p = 113$, matrix ≈ 6 times smaller.

Main optimizations (6) – Truncation

- Most of our computations are done in high precision $\Rightarrow$ we have to reduce large matrices ;
- Little superfluous precision : reducing it means discarding some of the remainders ;
- Replacing $F_{[b,2b-a]}$ by $\texttt{trunc}(2^A F_{[b,2b-a]})/2^A$ works very poorly;
- However, replacing $U_{[a,b]}F_{[b,2b-a]}$ by $\texttt{trunc}(2^A U_{[a,b]}F_{[b,2b-a]})/2^A$ works very well ;
- ... we can reduce the precision by a factor of ≈ 6.
- $\Rightarrow$ gain up to a (disappointing) factor 2 in practice.

Implementation

Only a few words:

- Two implementations;
- One in sage (1D + 2D);
- One in C (2D, multithreaded);
- in the 2D version the C code is better;
- Both ≈ 8 kloc ;
- Use Arb (ball arithmetic) + fpLLL (not the one bundled in sage).
- Specific code for $d = 2$, $(N_1, N_2) = (5, 2)$ for the "classical" TMD $(w = 2)$.

Experiments

- For $p = 113$, $f = \exp$, over $[1/4, 1/2]$
- CPU = Xeon E5620 @2.4 GHz (my laptop = twice faster).

d	w	int. wdth	Time	% matrix	% LLL	Prered (gain on LLL)	Trunc (gain on LLL)
6	6	$2^{-32.1}$	0.06s	20%	79%	/ 4.3	20%
12	6	$2^{-24.2}$	8.7s	2.7%	97.7%	/ 12	50%
8	8	2^{-25}	0.47s	12%	87%	/ 9	27%
12	8	2^{-21}	9.4s	4.4%	95.6%	/ 10	40%
10	10	$2^{-20.2}$	3.8s	6.3%	93.7%	/ 10	31%
12	10	$2^{-19.3}$	8.7s	5.2%	94.7%	/ 13	44%
12	12	$2^{-16.8}$	12.8s	4.6%	95.4%	/ 14	

Conclusion

- Using the function (via a $P + r$ representation) rather than only P has a major impact;
- Turning this into a formal proof is going to be somewhat harder than SLZ;
- Results are quite encouraging for large w;
- Preliminary results in small degree suggest that we might compete, to some extent, with Lefèvre's implementation of his algorithm;
- Further (significant) progress will imho require going into LLL internals.